COMP26120: Algorithms and Imperative Programming

Lecture 3:
Control flow
Information representation (part I)
Lecture outline

- **Control flow**
  - Sequencing;
  - Selection (if, nested if, chained if, switch);
  - Iteration (for loops, while and do while loops);

- **Information representation**
  - Introduction to memory and types;
  - Type conversion and casting;
  - Pointers;
  - Memory management and allocation;
Control flow Sequencing

- Performing an activity frequently consists of a number of sub-activities, which need to be performed correctly and in a correct sequence.
- Dijkstra identified sequencing in computer programming as one of the three control constructs. An important feature of a sequence of instructions is that, for the same input parameters, the sequence will always yield the same result.
- **Example:**

```c
#include <stdio.h>

int main()
{
    scanf("...");
    printf("...");
    return 0;
}
```
Control flow
Selection

- The second control construct identified by Dijkstra is the selection.
- **Example**: If I wake up, I will go to the COMP26120 lecture, else I will stay in bed.

```
if <condition> then
    {do something}
else
    {do something else}
```

- Conditions are formed using **relational operators**. These compare two values and return a true/false answer which controls further flow of execution.
- Some examples of relational operators include: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less or equal), `>=` (greater or equal), and `!=` (not equal to).
Control flow

Multiple selection

- If within the body of `then` or `else` branch another `if` statement(s) exist, we have a nested if structure.

- Example:

```plaintext
if (condition) then
{
    if (condition) then
    {
        do something
    }
    else
    {
        do something else
    }
}
else
{
    do this
};
```
Control flow
Multiple selection

- A multiple choice construct can be created using multiple `elseif` constructs.

- **Example:**
  ```
  if (condition1) then 
  {do this 1}
  else if (condition2) then 
  {do this 2}
  else if (condition3) then …
  …
  else
  {do this n}
  ```

- Not recommended to have too many branches (affects the performance of execution).
Control flow
Multiple selection

- An cleaner and more effective alternative to the chained `if ... then ... else ...` construct is available in many programming languages.
- In C this is a switch construct.
- **Example:**
  ```c
  switch (integer expression)
  {
  case constant1:
    {do this 1};
  case constant2:
    {do this 2};
  ...
  default:
    {do this default};
  }
  ```
Control flow
Iteration

- This makes the third form of program control (Dijkstra).
- It is used when a certain number of statements need to be repeated fixed or variable number of times (i.e. until a certain condition is satisfied).
- C has three different types of loops: **for** loop, **while** loop and **do-while** loop.

```c
for (initialisation, exit criterion, update) {
   do this
}

while (condition) {
   do this
}

do {
   this
} while (this condition is true);
```

```c
for (int i=0, i<10, i++) {
   printf('%3u', i);
}
```

```c
i=0;
while (i<10) {
   printf('%3u', i);
   i++
}
```

```c
i=0;
do {
   printf('%3u', i);
   i++
} while (i<10);
```
Control flow
Recursion

- There is an alternative way to explicit loop constructs to execute repeatedly some part of the code. This method is based on functions that call themselves.

- **Homework**: Read on functions in C (important for program structuring !)
  

- In order to understand recursion, you must first understand recursion.

- **Example**: Fibonacci sequence

  \[ a_{\downarrow 0} = 0, \quad a_{\downarrow 1} = 1, \quad a_{\downarrow n} = a_{\downarrow n-1} + a_{\downarrow n-2}, n \geq 2 \]

  ```c
  unsigned int fib(unsigned int n)
  {
      /* Base case 1: n = 0, so return 0 */
      if(n == 0) return 0;
      /* Base case 2: n = 1, so return 1 */
      else if(n == 1) return 1;
      /* Recursive case: n >= 2, return the result of previous fib()s */
      return fib(n - 1) + fib(n - 2);
  }
  ```
Control flow
Exiting the loop

- In some cases it is necessary to escape from the current block of code either completely, or to move directly to the next iteration within a loop.
- In the C language, the `break` statement to make an early exit from `for`, `while`, `do...while` loops.
- The `continue` statement, used in conjunction with `for`, `while` and `do...while` causes the next iteration of the loop to begin, skipping any remaining code within the loop.

```c
main( )
{
    int num, i;
    printf(“Enter a number”);
    scanf(“%d”, &num); i = 2
    while (i <= num - 1)
    {
        if (num%i == 0)
        {
            printf(“Not a prime number”);
            break;
        }
    }
}
```

```c
main( )
{
    int i,j;
    for(i = 1; i <= 2; i++)
    {
        for(j=1; j<=2; j++)
        {
            if (i==j) continue;
            printf("%d%d", i,j);
        }
    }
}
```
Information representation
Memory and types

- Main memory consists of the physical and virtual memory available to the operating system.
- When a program is loaded into main memory, a sufficient space to hold the program instructions and program data is allocated.
- How much memory do we assign to the individual data variables in the program? The answer: introduce data types.
- Using data types allows to define the amount of memory needed to represent a data item.
- Another important reason for using data types is to allow the compiler checks of a program for consistency.
Information representation
Basic data types

- C provides a standard, minimal set of basic (primitive) data types. More complex data structures can be built up from these basic types.

- **Integer types:**
  - **char** ASCII character at least 8 bits, which is sufficient to store a single ASCII character.
  - **short** has at least 16 bits which provides a signed range of integers \([-32768, 32767]\).
  - **int** is a default integer, typically 32 bits long, supporting the range of numbers \([-2 \text{ billion}, +2 \text{ billion}]\).
  - **long** Large integer, either 32 or 64 bites long (depending on a compiler).
Information representation
Basic data types

• Floating point types.
  ◦ float is a single precision floating point number of typical size 32 bits.
  ◦ double is a double precision floating point number of typical size 64 bits.
  ◦ long double is a possibly even bigger floating point number (compiler dependent).
  ◦ Constants in the source code such as 3.14 are default to type double unless the
    are suffixed with an ‘f’ (float) or ‘l’ (long double).
  ◦ Single precision (float) gives about 6 digits of precision and double about 15.

• Mathematical operators.
  ◦ + Addition
    ◦ — Subtraction
  ◦ / Division
  ◦ * Multiplication
  ◦ % Remainder (mod)

• Unitary increment operators. They pre/post increment/decrement the variable.
  ◦ ++ or — —
Information representation
Basic data types

- **Boolean type.**
  - There is no Boolean type – use `int` and constants 0 and 1.
  - The value 0 is false, anything else is true. The operators evaluate left to right and stop as soon as the truth or falsity of the expression can be deduced (short cutting).
  - Boolean type is equipped with logical operations (`! not (unary), `&&` and `|| or`).

- **Bitwise operators.**
  - C includes operators to manipulate information at the bit level. This is useful for writing low-level hardware or operating system code.
  - The user needs to check whether the bit manipulation code works correctly on a given architecture. The bitwise operations are typically used with unsigned types.
  - Bitwise operators are: `~` bitwise negation (unary), `&` bitwise and, `|` bitwise or, `^` bitwise exclusive or, `>>` right shift by right hand side (RHS) (divide by power of 2), `<<` left shift by RHS (multiply by power of 2).
Information representation
Variable definitions

- Every variable must be defined by its type prior to its first use.
- **Examples:**
  ```
  short number;
  char letter;
  float fraction;
  const double pi;
  for(int=0; i<10; i++)
  ```
- In the above definitions the memory location of a prescribed size is allocated for a variable, but its value can be arbitrary.
  ```
  short number=1;
  char letter=‘n’;
  float fraction=100.2;
  const double pi=3.1416;
  ```
Information representation

Type conversions

- When an expression contains the operators of different (but compatible) types, C may perform an implicit type conversion. This means it may treat the operand as though it were a type suitable for the operator.

- Example:

  ```c
  int lower (int c)
  {
    if (c >= 'A' && c <= 'Z') return c + 'a' - 'A';
    else return c;
  }
  ```

- The above function is correct, since chars are actually short integers (with the values -128 to 127).

- The implicit conversions of types in an expression should not lead to the loss of information:

  ```c
  int i= 123;
  float f;
  f= i;
  ```
Information representation
Type conversions and casting

- Example:

  int x = 5;
  int y = 6;
  y + x;

  short x = 5;
  int y = 6;
  y + x;

- The example on the right is a **mixed expression**.
- In this expression the value of x (a **short**) is converted to an **int**:
  1. x is copied into a register and converted into an **int**;
  2. y is copied into a register;
  3. The registers are added to give an **int** result
- Note that the value of x as stored in memory is **not** changed. Only the temporary copy is converted during the computation of the expression's value.
The process of converting operands of an arithmetic expression to a different type in order to evaluate an expression is referred to as **implicit conversion**. The rules for determining the type of a mixed arithmetic expression are:

1. Operands of type `char` or `short` are converted to an `int`. (`unsigned char` or `unsigned short` is converted to `unsigned int`).
2. If the expression is still of a mixed type, the operand of lower type is converted to that of the higher type operand according to:

   ```
   int < unsigned < long < unsigned long < float < double
   ```

The result is always of the higher type.
Information representation
Type conversions and casting

- Implicit conversion occurs across assignment and in mixed arithmetic expressions.
- In addition to this, a programmer can state explicitly the type of a variable or value. This is known as casting or explicit conversion.
- Example: If ch is of type char then (int)ch will cast the value stored in ch as an int. Note that the value of ch in memory will still be a char and that only a copy is cast.
- Example: Casting of an expression:
  
  (double)(6 * x +4.6)
  
  (int)(f * 3.142)

- The casting operator is a unary operator, thus it has the same precedence as other unary operators.
Homework. Find the meaning of the following expressions (yes, they are all legal in C):

- `i%5;`
- `a = (b = (c = 5));`
- `x = (a<b) ? a : b;`
You have met these already in Java (and ARM — recall indirect addressing).

Pointers need to be declared as any other variable:

```c
int a;
int *a_ptr;
a=567;
a_ptr=&a;
```

The operator & assigns the address to a pointer.
Using the address stored in a pointer is called **dereferencing a pointer** and is achieved by the operator `*`. When applied to a pointer variable, it accesses the data the pointer is referring to.

```c
int a;
int *a_ptr;
a = 567;
a_ptr = &a;
a = 890; (alternatively *a_ptr = 890)
```

Notice that the value of `a` has changed to 890, but the value of the pointer `a_ptr` is still `0x123`. 
Information representation

Pointers

• In C, a pointer can point to anything, even to another pointer, for example:

```c
int a;
int *a_ptr = &a;
int **aa_ptr = &a_ptr;
**aa_ptr = 220;
```

• What is the value of *a_ptr and what is the value of **aa_ptr?

```c
int i = 5, j = 6; k = 7;
int *ip1 = &i, *ip2 = &j, **ipp=&ip1;
*ipp=ip2;  (?)
*ipp=&k;   (?)
```
Information representation

Pointers

- **Example:** Remember the `scanf()` function?
  ```c
  char c1;
  char c2;
  scanf("%c %c", &c1, &c2);
  ```

- Remember also that you need to read about the functions?

- The main purpose of functions is to structure the program. Functions are often passed some data to operate on and will often return data.

- In your tutorial you have had a question why do we need to pass the addresses to the char variables c1 and c2, rather than the variables themselves?
Information representation

Pointers

- By sending the addresses to `scanf` we let the function “know” where to store the values it has scanned in, i.e. it needs to change the actual memory locations where these values are stored.
- We say that `scanf` requires the arguments to be passed by reference, and thus it expects to be sent arguments of type pointer, which are addresses. This is achieved by using the `&` operator on `n`.
- We could have sent a pointer that contained the address of `n` instead, for example:

```c
char c1;
char c2;
char *ptr_c1;
char *ptr_c2;
ptr_c1=&c1;
ptr_c2=&c2;
scanf(“%c %c”, ptr_c1, ptr_c2);
```
Information representation

Pointers

- This is an instance of the problem of passing the arguments to a function by reference vs. passing them by value.
- If you use the address of a variable, you pass this variable by reference. This means that whenever one changes the variable within the function, this change will be registered in the memory.
- When passing a variable by value, a separate copy of this variable is created and whatever changes in the function are made to that copy will not affect the value of an original variable.
- In scanf() function we need to change the value of a variable itself, and hence we need to pass it by reference (i.e. pass its address).
Information representation
Memory allocation

- Until now we know in advance the amount of memory that we need for each of the variables in a program. But, is this always the case? Think of an example…
- If not, how can we make the code to allocate/free the memory dynamically (on demand)?
- In C this can be achieved by the following functions:
  - `sizeof(type)` This function returns the type size (in bytes). Remember that the number of bytes for various types can be machine/compiler dependent.
  - `malloc(size)` This function allocates a storage of a certain size. It returns a pointer to this storage.
  - `free(pointer)` This function will free a part of memory previously allocated with `malloc`. The input argument is the pointer to the memory to free.
Information representation

The sizeof() function

- Before a dynamical allocation of memory we need to know how much memory different types of data take up.
- The sizeof() function can perform this task.
- **Example:**
  
  ```
  sizeof (int)
  sizeof (char)
  sizeof (double)
  ```
- The function returns an integer that represents the number of bytes used to store an object of that type.
Information representation

The malloc() function

• The memory allocation is performed by the function `malloc()` (which means memory allocation).

• The argument of `malloc` is an integer argument which is the number of bytes of memory that we want. The memory is allocated from the heap and a pointer to that block of memory is returned.

• If there is insufficient storage in the heap, `malloc` returns `NULL`.

• The pointer returned when `malloc` is successful is 'untyped' (it is a pointer of type `void *`). It needs to be casted to a particular type before its use.

• Example:

```c
char *new_memory;
int memory_size = 20 * sizeof(char);
new_memory = (char *)malloc(memory_size);
if(new_memory == NULL)
    printf("No memory allocated!\n");
else
    printf("Memory for 20 characters allocated\n");
```

• **Homework:** Extend this program to write 2 chars at the start of this block.
Information representation
The free() function

- Good programming practice is to free the allocated memory when we no longer need it (C does not have automatic garbage collection!).
- The argument of `free()` function is a pointer to the beginning of a block that you have been allocated using the `malloc` function. (need to be preserved between the calls to `malloc` and `free`, or make a copy of it).
- In the previous example `free(new_memory)` will release the allocated block.
- **Homework**: How would you free the memory after doing the homework from the previous slide?